Unidad II: Programación orientada a objetos básica

Lic. Ronaldo Armando Canizales Turcios

Departamento de Electrónica e Informática, UCA

Ciclo virtual 01-2021

Agenda

1. **Cohesión y acoplamiento**
2. **Sobrecarga de métodos**
3. **Sobrecarga de constructores**
4. **Static keyword**
   - Métodos estáticos
   - Atributos estáticos
   - Clases estáticas

1. Cohesión y acoplamiento

En la presente sección se abordarán dos conceptos útiles para analizar la calidad de la interfaz pública de una clase. **Una clase debe representar un único concepto.** Los métodos y atributos que forman parte de la interfaz pública deben ser **cohesivos**. Esto significa que todo lo que pertenezca a la interfaz pública deberá estar estrechamente relacionado con el único concepto que dicha clase representa.

cohesión

Der. del lat. *cohaesus*, part. pas. de *cohaerēre* 'estar adherido', 'tener cohesión'.

1. f. Acción y efecto de reunirse o adherirse las cosas entre sí o la materia de que están formadas.
2. f. **enlace** (‖ unión de algo con otra cosa).
3. f. *Fís.* Unión entre las moléculas de un cuerpo.
4. f. *Fís.* Fuerza de atracción que mantiene unidas las moléculas de un cuerpo.

1. Cohesión y acoplamiento

Si la interfaz pública de una clase se refiere a múltiplos conceptos, entonces es una buena señal de que se debería **separar** dicha clase en varias clases. Por ejemplo: considere la interfaz pública de la siguiente clase Monedero.

Definición de una clase que no tiene cohesión

```
public class Monedero {
    public Monedero() {...}
    public void agregarCentavos(int cantidad) {...}
    public void agregarCoras(int cantidad) {...}
    public void agregarDolares(int cantidad) {...}
    public double consultarTotal() {...}

    private double valorCentavo, valorCora, valorDolar;
    private int cantCentavos, cantCoras, cantDolares;
}
```

Versión 1

1. Cohesión y acoplamiento

Aquí coexisten dos conceptos:
1) Un monedero que contiene monedas y calcula el total.
2) El valor de cada moneda en específico.

Tendría más sentido separar dicha clase en una clase **Moneda** que sea responsable de almacenar su nombre y su valor.

Interfaz pública de la clase Moneda

```
public class Moneda {
    public Moneda(double pValor, string pNombre) {...}
    public double getValor() {...}
    public double getNombre() {...}
    ...
}
```

Versión 2

1. Cohesión y acoplamiento

De esta manera, la clase **Monedero** puede ser simplificada de la siguiente manera:

Interfaz pública de la clase Monedero

```
public class Monedero {
    public Monedero() {...}
    public void añadirMoneda(Moneda pMoneda) {...}
    public double consultarTotal() {...}
    ...
}
```

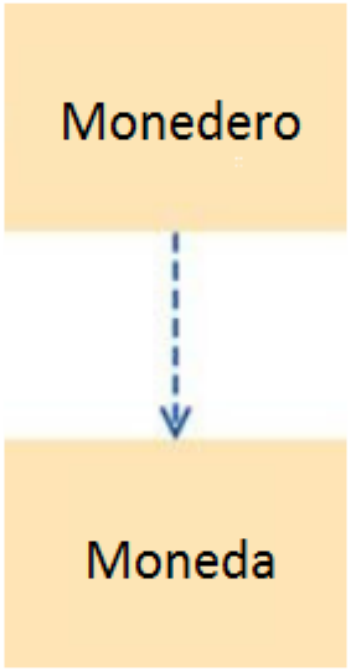
Versión 2

Este diseño es mejor que el anterior pues *tiene mayor cohesión* (separa las responsabilidades de un monedero y de las monedas).

1. Cohesión y acoplamiento

Muchas clases necesitan otras para poder realizar su trabajo.

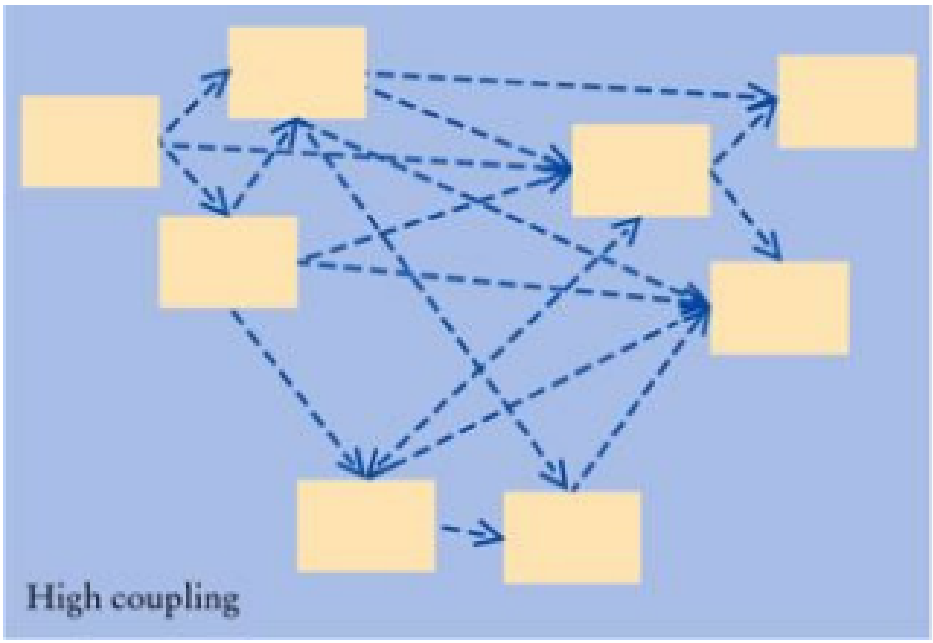
Por ejemplo, la clase Monedero ahora **depende** de la clase Moneda para calcular el total de dinero que almacena. Podemos visualizar la relación de dependencia entre las clases Moneda y Monedero de la siguiente manera:

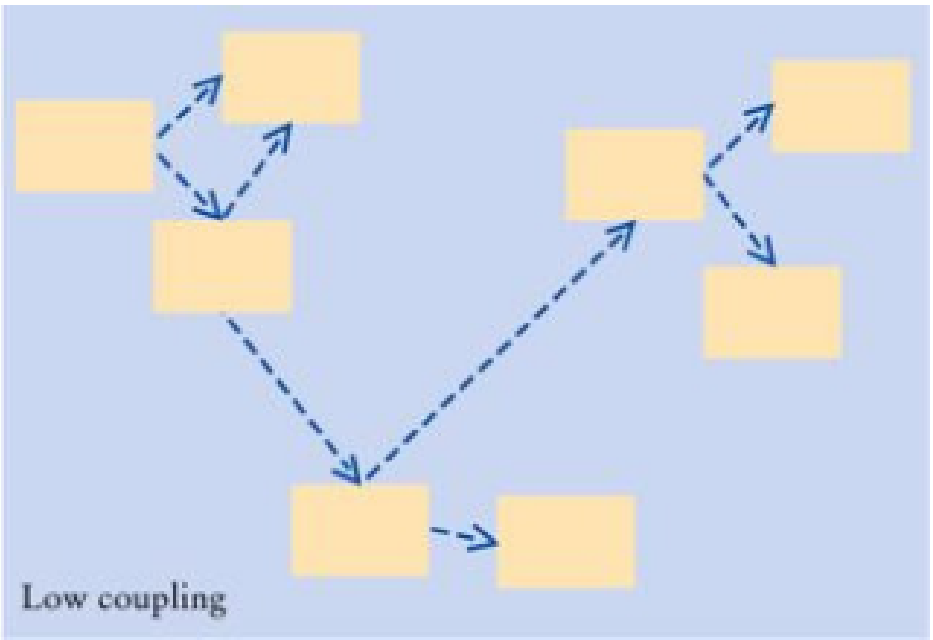


Nótese que la clase **Moneda** no depende de la clase **Monedero**. Las monedas no tienen noción de que están siendo recolectadas en monederos, las monedas existen independientemente de si son almacenadas en un monedero o no.

1. Cohesión y acoplamiento

Si en un programa existen muchas clases que dependen unas de otras, entonces se dice que el **acoplamiento** entre clases es alto. El caso contrario también aplica, si hay poca dependencia entre clases, entonces se dice que el **acoplamiento** es bajo.





1. Cohesión y acoplamiento

¿Porqué importa poner atención al acoplamiento en nuestros programas?

- Si la clase **Moneda** cambia en alguna versión futura, entonces todas las clases que dependen de ella podrían verse afectadas. Si el cambio es grande, entonces todas las clases *acopladas* deben ser actualizadas también.
- Si en un futuro quisiéramos utilizar una clase en otro programa, entonces tenemos que llevar también todas aquellas clases que están *acopladas* a ella. Es decir, si queremos utilizar la clase **Monedero** en otro programa, entonces tenemos que llevar también a la clase **Moneda**.

Como conclusión, de ahora en adelante será importante:

- Maximizar la cohesión de la interfaz pública de nuestras clases.
- Minimizar el acoplamiento innecesario entre clases.

2. Sobrecarga de métodos

Cuando se utiliza el mismo nombre para más de un método o constructor, entonces se dice que dicho nombre está **sobrecargado**. Esto es particularmente común en los constructores, porque todos los constructores deben tener el mismo nombre (el nombre de la clase).

En C# y en Java es posible **sobrecargar** métodos y constructores **siempre y cuando los parámetros sean distintos**, es decir, distinta cantidad de parámetros o distinto tipo.

Para sobrecargar métodos, importa la cantidad y el tipo de los parámetros que se piden, pero **no importa el tipo de dato devuelto por el método**. Es decir que **no es posible** crear dos métodos que tengan el mismo nombre, parámetros idénticos (en cantidad y tipo) pero que retornen valores distintos.

2. Sobrecarga de métodos

Ejemplo:

Sobrecargue el método **añadirMoneda** de la clase Monedero. De tal modo que en lugar de pedir una Moneda, se pueda pedir el valor y el nombre de la moneda que se quiera introducir.

```
public void añadirMoneda(Moneda unaMoneda) {
    listaMonedas.Add(unaMoneda);
}
public void añadirMoneda(double unValor, string unNombre) {
    Moneda unaMoneda = new Moneda(unValor, unNombre);

    listaMonedas.Add(unaMoneda);
}
```

Versión 3

3. Sobrecarga de constructores

Llamar a un constructor desde otro constructor

Respecto de la clase **Moneda**: ¿qué sucedería si quisiéramos implementar dos constructores?

- Uno que solicite los parámetros de construcción pValor y pNombre.
- Otro que no solicite ningún parámetro de construcción, pero que *por defecto* cree una moneda de a dólar (Nombre: Un dólar. Valor = \$1).

Claro, es posible inicializar el atributo nombre con la cadena de texto "*Un dólar*" y el atributo valor con \$1. Pero también es posible **llamar a otro constructor de la misma clase** y proveerle dichos valores.

3. Sobrecarga de constructores

Llamada a un constructor desde otro en C#

```
public Moneda(double unValor, string unNombre) {
    valor = unValor;
    nombre = unNombre;
}
```

Versión 3

```
public Moneda() : this(1, "Un dólar") {
    //Otras sentencias, si se desea (opcional)
}
```

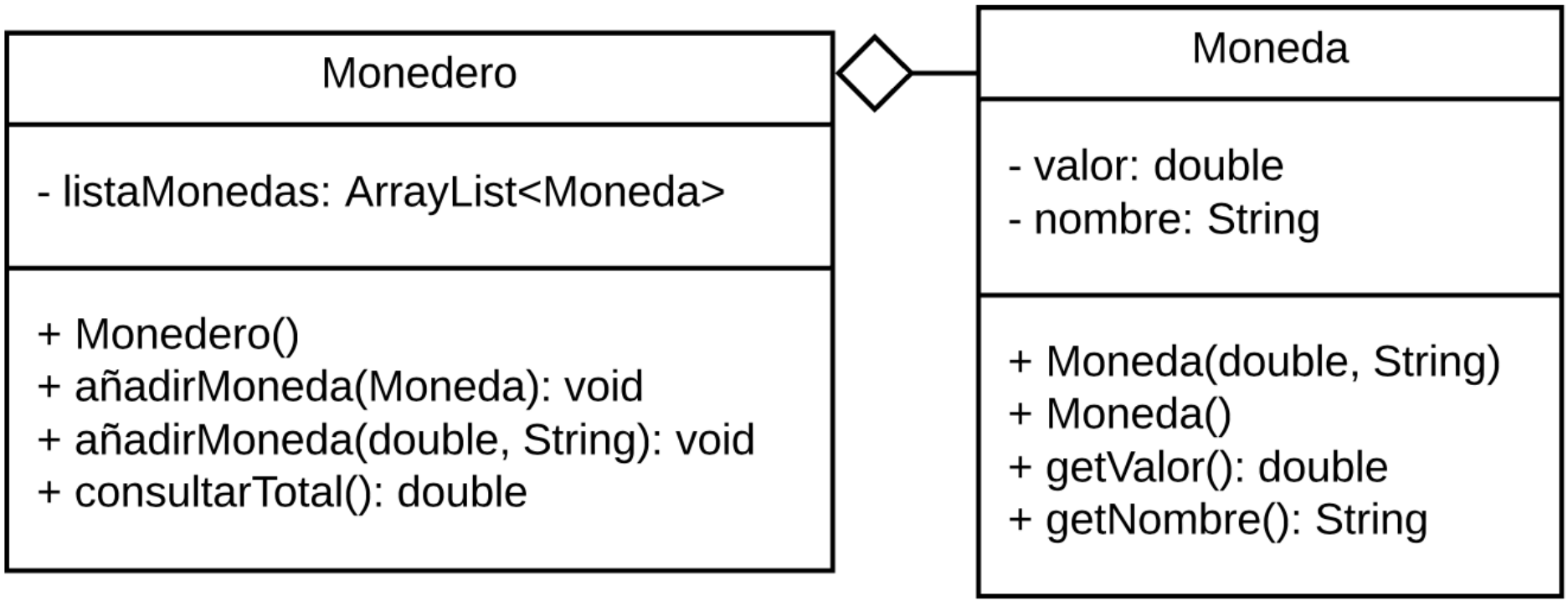
Versión 4

La sintaxis es la que se muestra en el cuadro anterior. **Importante:** se debe utilizar la palabra reservada **this**.

3. Sobrecarga de constructores

Ejemplo:

Realizar diagrama UML de clase para Moneda y Monedero.



Métodos, atributos y clases estáticas

Aclaración: aunque la traducción pudiese ser engañosa, “static” NO significa estático ni constante. Significa que el miembro **pertenece a la clase** y no a una instancia en específico.

Static tampoco significa que algo esté fijo o no pueda cambiar; **sí podrá cambiar** pero tendrá el mismo valor para todos los objetos que se hayan instanciado de una clase en particular.

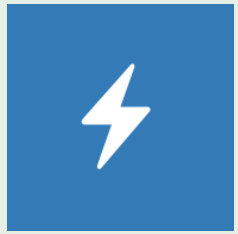
4.1 Static keyword: Métodos estáticos

Es posible decir que los métodos se clasifican en:

- **Métodos de instancia:** son aquellos que operan en una instancia específica de una clase. Por ejemplo, todos los que hemos implementado hasta el momento.
- **Métodos de clase:** también llamados *métodos estáticos*, Por el contrario, pertenecen a una clase pero no a una instancia en específico. Para utilizarlos NO es necesario instanciar ningún objeto.

Ejemplo: el método pow de la clase Math

double a = 30;
double b = 2;
*double c = Math.***Pow***(a, b);*



4.1 Static keyword: Métodos estáticos

Una clase **no estática** puede contener métodos o atributos estáticos. El miembro estático es invocable[a] en una clase, incluso si no se ha creado ninguna instancia de la clase. Siempre se tiene acceso al miembro estático **con el nombre de clase**, no con el nombre de instancia.

[a] Se puede utilizar desde el Main.

Solo existe **una copia** de un atributo estático, independientemente del número de instancias (objetos) de la clase que se creen.

Inglés con subtítulos

Los métodos estáticos **no pueden** tener acceso a atributos no estáticos, y tampoco pueden tener acceso a una variable de instancia de un objeto a menos que se pase explícitamente en un parámetro de método.

4.2 Static keyword: Atributos estáticos

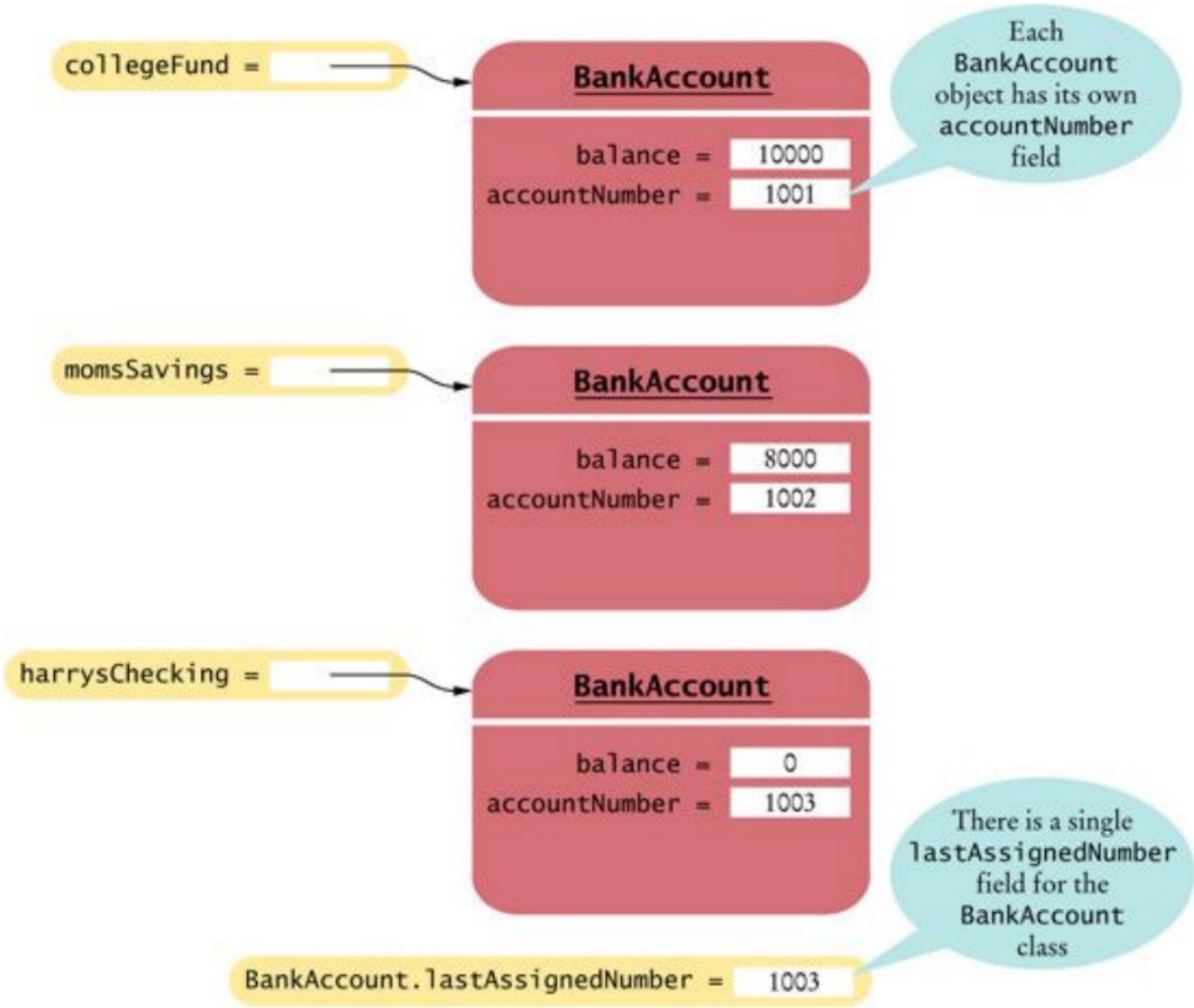
¿Cómo implementaría usted el siguiente requerimiento?

Se desea agregar el atributo *número de cuenta* a la clase **CuentaBancaria**. También se desea que la asignación del número de cuenta se dé de forma secuencial. Es decir, la primera vez que se instancie un objeto de la clase CuentaBancaria deberá tener 1 como número de cuenta; la segunda vez deberá de tenerse 2 como número de cuenta, y así sucesivamente;

De esta forma, cada cuenta tendrá un número asignado de manera automática y será imposible que existan duplicados.

4.2 Static keyword: Atributos estáticos

La pregunta del millón: ¿un atributo llamado *último número de cuenta asignado* pertenecería a una instancia en específico o a la clase como tal?



4.2 Static keyword: Atributos estáticos

Existen tres formas de inicializar atributos estáticos:

- No hacer nada. Entonces los números se inicializarán con cero, los valores booleanos con falso y las variables de tipo objeto con null.
- Utilizar un inicializador explícito.

Es importante mencionar que una buena práctica de programación es: **no abusar en el uso de atributos estáticos**.

4.3 Static keyword: Clases estáticas

Una **clase estática** es aquella clase que se usa sin necesidad de realizar una instanciación de la misma. Se utiliza como una unidad de organización para métodos no asociados a objetos particulares y separa datos y comportamientos que son independientes de cualquier identidad del objeto.

- Se usa el modificador **static** para definir una clase estática.
- La clase no debe implementar ningún interfaz ya que que los métodos del interfaz son llamados en una instancia de la clase.
- La clase debe definir únicamente miembros estáticos. Cualquier miembro de instancia provocará un error de compilación.
- La clase no puede usarse como un campo, como parámetro de un método o como variable local ya que todos estos conceptos refieren a una instancia.

Las clases estáticas son adecuadas cuando no tienen que almacenar información, sino sólo realizar cálculos o algún proceso que no cambie.

4.3 Static keyword: Clases estáticas

Características principales de una clase estática

- Solo contiene miembros estáticos.
- No se pueden crear instancias de ella.
- No puede contener constructores de instancias.
- No se utiliza el operador **new**.
- Dado que no hay ninguna variable de instancia, para tener acceso a los miembros de una clase estática, debe usar el nombre de la clase.

Por lo tanto, crear una clase estática es básicamente lo mismo que crear una clase "normal" que contiene solo miembros estáticos.

4.3 Static keyword: Clases estáticas

Es posible usar una clase estática como un contenedor adecuado para conjuntos de métodos que solo funcionan en parámetros de entrada y que no tienen que obtener ni establecer campos de instancias internas.

Ejemplo

Leer y anexar datos a un archivo de texto.

Patrón de diseño Singleton

Las clases estáticas presentan muchas similitudes (y diferencias) con el patrón de diseño Singleton, del que hablaremos más adelante en la materia.

4.3 Static keyword: Clases estáticas

Ejemplo

La solución propuesta al problema de la **generación automática** de un ID para cuentas bancarias que se describió en la sección pasada (diapositivas 15 - 16) funciona bastante bien, pero pierde un poco de **cohesión**[a] ya que no debería ser 100 % responsabilidad de la clase CuentaBancaria el llevar registro de cuántas cuentas ha abierto el banco.

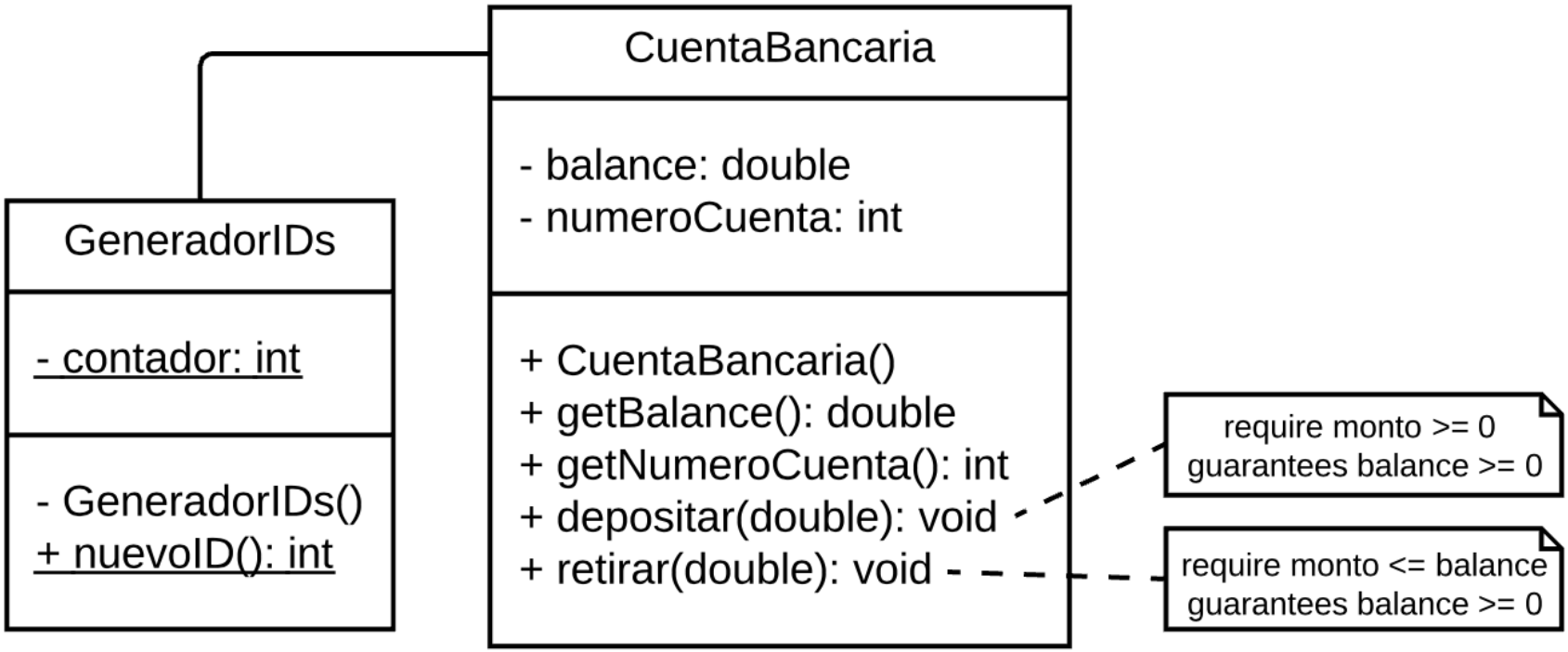
[a] Principio de responsabilidad única.

Lo ideal sería **separar esa responsabilidad** a otra clase, que se encargue exclusivamente de eso. Y qué mejor que hacerlo con una **clase estática**.

4.3 Static keyword: Clases estáticas

Ejemplo

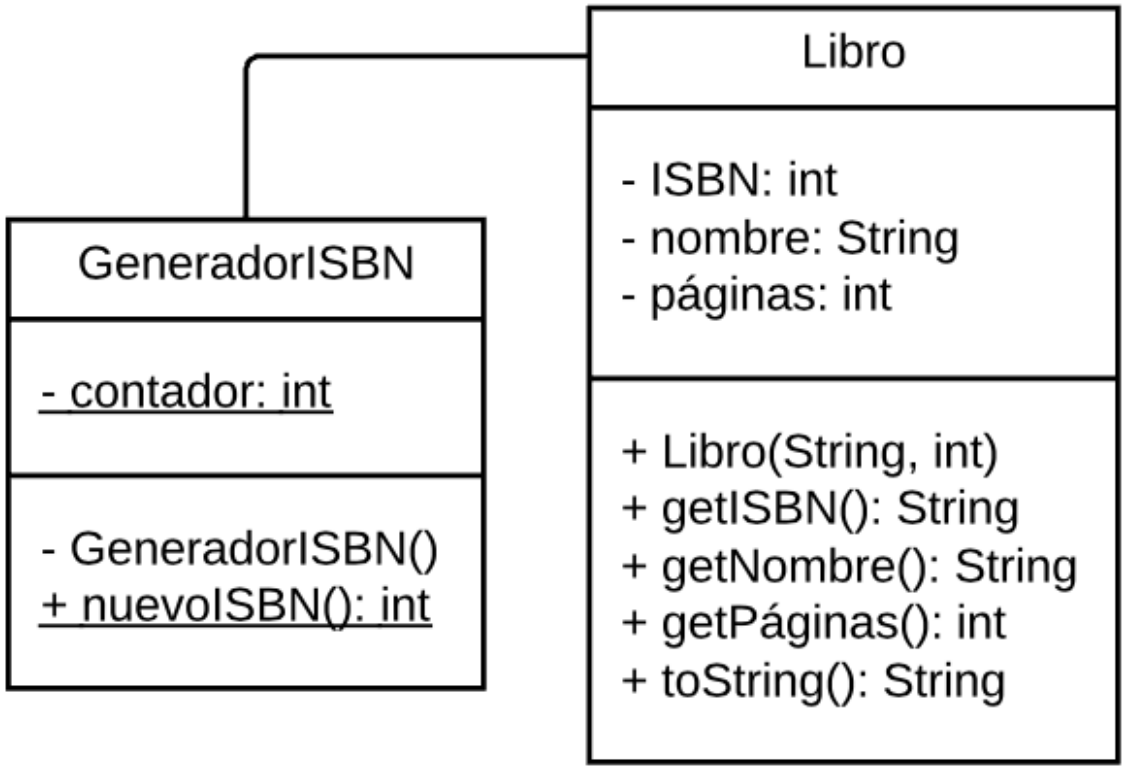
Realizar el diagrama UML de clases del programa descrito en la diapositiva anterior.



4. Static keyword

Ejercicio sugerido (no ponderado)

Implementar la generación automática del código ISBN de cada libro mediante una clase **estática**, tomando como base el siguiente diagrama UML.



Unidad II: Programación orientada a objetos básica

Lic. Ronaldo Armando Canizales Turcios

Departamento de Electrónica e Informática, UCA

Ciclo virtual 01-2021